M250 Object-oriented Java programming

M250 Exam Handbook

The Open University

Contents

Printed in the United Kingdom

SUP 04180 4
2.1

1 Introduction

This booklet provides a quick reference to most of the classes and interfaces you have encountered in M250 and is intended for use during your study of this module and in the final exam. It is based on the Javadoc for these classes and interfaces, but differs from the Javadoc in the following three respects.

- For the M250 example classes `private` instance variables *are* shown.
- In Sections 4, 5 and 6 we have simplified the class and method comments and, in places, the method headings.
- For each class or interface we do not necessarily show every method that is available. If you want more detail you can access the full documentation for the Java Class Libraries from BlueJ's `Help` menu.
- Similarly, we do not always list any or all of the interfaces implemented by a class.

This booklet is not designed to be read from cover to cover; rather you should use its index to find the documentation for a particular method, class or interface.

1.1 Use of the M250 Exam Handbook in the examination

You will be allowed to take this booklet into the examination, however the following conditions apply.

- You can *only* take the version of this booklet that was printed and sent to you by The Open University.
- Basic annotation as described in the Examination Arrangements booklet is permitted. This means that the text as printed may be supplemented by handwritten highlights (for example by a highlighter pen or by ringing, underlining or sidelining), and by corrected typographical errors. The addition of comments, marginal notes, notes in the blank spaces at the end of paragraphs and pages, or on fly-leaves is ***not*** permitted.

The PDF for this booklet is available on the M250 website, so you are free to print your own copy of this booklet and annotate it how you wish, but remember such a copy *cannot* be taken into the examination.

2 OU Class Library

Class `OUAnimatedObject`

```
java.lang.Object
  └ java.util.Observable
      └ ou.OUAnimatedObject
```

`public abstract class OUAnimatedObject extends Observable`

`OUAnimatedObject` is the superclass (either direct or indirect) of all classes in M250 whose objects have a representation in the `Graphical Display`.

Method Summary

`void performAction(String action)`

Notifies all observers of the `OUAnimatedObject` that it has performed the action defined by `action`.

`void update(String instanceVariable)`

Notifies all observers of the `OUAnimatedObject` that the instance variable corresponding to `instanceVariable` has been changed.

Class `OUColour`

```
java.lang.Object
  └ java.awt.Color
      └ ou.OUColour
```

`public class OUColour extends Color`

Extends `java.awt.Color` to include an improved `toString()` method and the static colours `BROWN` and `PURPLE`.

Instance variables

`static OUColour BLACK`

The colour black.

`static OUColour BLUE`

The colour blue.

`static OUColour BROWN`

The colour brown.

`static OUColour CYAN`

The colour cyan.

`static OUColour GREEN`

The colour green.

`static OUColour MAGENTA`

The colour magenta.

`static OUColour ORANGE`

The colour orange.

`static OUColour PINK`

The colour pink.

`static OUColour PURPLE`

The colour purple.

`static OUColour RED`

The colour red.

`static OUColour WHITE`

The colour white.

`static OUColour YELLOW`

The colour yellow.

Constructor Summary

`OUColour(int r, int g, int b)`

Initialises the `OUColour` object with the specified red, green, and blue values in the range (0–255).

Method Summary

`String toString()`

Returns the string representing the colour.

Class `OUDialog`

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window
              └ java.awt.Dialog
                  └ javax.swing.JDialog
                      └ ou.OUDialog
```

`public class OUDialog extends JDialog`

`OUDialog` provides static methods to create dialogue boxes, either to display results, or request information.

Method Summary

`static void alert(String prompt)`

Displays a dialogue box with the message `prompt` and an `OK` button.

`static boolean confirm(String prompt)`

Displays a `Yes`/`No` dialogue box with the question `prompt`.

`static String request(String prompt)`

Displays a dialogue box with the question `prompt`, `OK` and `Cancel` buttons, and an edit box for data entry.

```
static String request(String prompt,
                      String initialAnswer)
```

Displays a dialogue box with the question `prompt`, `OK` and `Cancel` buttons, and an edit box for data entry that contains the default answer `initialAnswer`.

Class `OUFileChooser`

```
java.lang.Object
    └ java.awt.Component
        └ java.awt.Container
            └ javax.swing.JComponent
                └ javax.swing.JFileChooser
                    └ ou.OUFileChooser
```

`public class OUFileChooser extends JFileChooser`

`OUFileChooser` provides static methods to allow the user to select a file for input or output.

Method Summary

`static String getFilename()`

Displays a file chooser dialogue with `OK` and `Cancel` buttons allowing the user to browse the file hierarchy and select a file. Returns a string representing the selected file's pathname.

`static String getFilename(String fileName)`

Returns a string representing the pathname of a file in the current folder with the name `fileName`.

`static void setPath(String path)`

Sets the current working directory to the folder specified by the `String` path.

3 M250 example classes

3.1 Amphibian classes

The amphibian classes documented in this section are the versions discussed in *Unit 6*, Subsection 5.1 and which first appear in Unit6_Project_14.

Various subclasses of amphibians such as `BovverFrog` are introduced in the module as a means of investigating and demonstrating various object-oriented concepts. We do not provide documentation for such classes here as they are only used briefly in a particular unit.

Class `Amphibian`

```
java.lang.Object
    ∟ java.util.Observable
        ∟ ou.OUAnimatedObject
            ∟ Amphibian
```

Direct known subclasses: `Frog`, `Toad`

`public abstract class Amphibian extends OUAnimatedObject`

The abstract class `Amphibian` is the superclass (either direct or indirect) of all M250 amphibian-like classes.

Instance variables

```
private OUColour colour
private int position
```

Method Summary

See also the superclass `OUAnimatedObject`.

`void brown()`

Sets the colour of the receiver to brown.

`void croak()`

Causes user interface to emit a sound.

`OUColour getColour()`

Returns the colour of the receiver.

`int getPosition()`

Returns the position of the receiver.

`void green()`

Sets the colour of the receiver to green.

`abstract void home()`

Resets the receiver to its ‘home’ position.

`abstract void left()`

Moves the receiver to the left.

`abstract void right()`

Moves the receiver to the right.

`void sameColourAs(Amphibian anAmphibian)`

Sets the colour of the receiver to the argument's colour.

`void setColour(OUColour aColour)`

Sets the colour of the receiver to the value of the argument `aColour`.

`void setPosition(int aPosition)`

Sets the position of the receiver to the value of the argument `aPosition`.

`String toString()`

Returns a string representation of the receiver.

Class `Frog`

```
java.lang.Object
    └ java.util.Observable
        └ ou.OUAnimatedObject
            └ Amphibian
                └ Frog
```

Direct known subclass: `HoverFrog`

`public class Frog extends Amphibian`

The class `Frog` defines an amphibian with the characteristics of a frog.

Constructor Summary

`Frog()`

Constructor for objects of class `Frog` that initialises `colour` to `OUColour.GREEN` and `position` to `1`.

Method Summary

See also the superclasses `Amphibian` and `OUAnimatedObject`.

`void home()`

Resets the receiver to its 'home' `position` of `1`.

`void jump()`

Causes a change in an appropriate observing user interface.

`void left()`

Decrements the `position` of the receiver by `1`.

`void right()`

Increments the `position` of the receiver by `1`.

Class `HoverFrog`

```
java.lang.Object
    └ java.util.Observable
        └ ou.OUAnimatedObject
            └ Amphibian
                └ Frog
                    └ HoverFrog
```

`public class HoverFrog extends Frog`

The class `HoverFrog` is a subclass of `Frog` with the additional instance variable `height` and an additional protocol to initialise and use `height`.

Instance variables

`private int height`

Constructor Summary

`HoverFrog()`

Constructor for objects of class `HoverFrog` that initialises `height` to `0`.

Method Summary

See also the superclasses `Frog`, `Amphibian` and `OUAnimatedObject`.

`void down()`

If the `height` of the receiver is greater than `0`, decrements the `height` of the receiver by `1`, otherwise the method does nothing.

`void downBy(int stepChange)`

Decreases the `height` of the receiver by the value of the argument `stepChange`. The new `height` of the receiver must lie in the range `0-6` otherwise the method does nothing.

`int getHeight()`

Returns the `height` of the receiver.

`void home()`

Resets the receiver to its ‘home’ `position` of `1` and to a `height` of `0`.

`void setHeight(int aHeight)`

Sets the `height` of the receiver to the value of the argument `aHeight`. `aHeight` must lie in the range `0-6` otherwise the method does nothing.

`String toString()`

Returns a string representation of the receiver.

`void up()`

If the `height` of the receiver is less than `6`, increments the `height` of the receiver by `1`, otherwise the method does nothing.

`void upBy(int stepChange)`

Increases the `height` of the receiver by the value of the argument `stepChange`. The new `height` of the receiver must lie in the range `0-6` otherwise the method does nothing.

Class `Toad`

```
java.lang.Object
    └ java.util.Observable
        └ ou.OUAnimatedObject
            └ Amphibian
                └ Toad
```

`public class Toad extends Amphibian`

The class `Toad` defines an amphibian with the characteristics of a toad.

Constructor Summary

`Toad()`

Constructor for objects of class `Toad` that initialises `colour` to `OUColour.BROWN` and `position` to `11`.

Method Summary

See also the superclasses `Amphibian` and `OUAnimatedObject`.

`void home()`

Resets the receiver to its 'home' `position` of `11`.

`void left()`

Decrements the `position` of the receiver by `2`.

`void right()`

Increments the `position` of the receiver by `2`.

3.2 Account classes

The `Account` class documented here is the version discussed in *Unit 6*, Subsection 3.2 and first seen in a complete form in Unit6_Project_4. The `CurrentAccount` class documented here is the version developed in *Unit 6*, Section 4 and first seen in a complete form in Unit6_Project_12_Sol.

Class `Account`

```
java.lang.Object
    └ Account
```

Direct known subclass: `CurrentAccount`

`public class Account extends Object`

The `Account` class models simple bank accounts, allowing money to be credited to, and debited and transferred from, an account.

Instance variables

```
private String holder
private String number
private double balance
```

Constructor Summary

`Account()`

Constructor for objects of class `Account`. Sets `holder` and `number` to empty strings and `balance` to `0.0`.

```
Account(String aHolder, String aNumber,
        double aBalance)
```

Constructor for objects of class `Account`, which sets the values of the `holder`, `number` and `balance` of the receiver to the arguments `aHolder`, `aNumber` and `aBalance` respectively.

Method Summary

`void credit(double anAmount)`

Credits the receiver with the value of the argument `anAmount`.

`boolean debit(double anAmount)`

If the `balance` of the receiver is equal to or greater than the argument `anAmount`, the `balance` of the receiver is debited by the argument `anAmount` and the method returns `true`, otherwise `false` is returned.

`boolean equals(Account anAccount)`

Returns `true` if the receiver is equivalent to (has the same state as) the argument `anAccount`, otherwise `false` is returned.

`double getBalance()`

Returns the `balance` of the receiver.

`String getHolder()`

Returns the `holder` of the receiver.

`String getNumber()`

Returns the `number` of the receiver.

`void setBalance(double anAmount)`

Sets the `balance` of the receiver to the value of the argument `anAmount`.

`void setHolder(String aHolder)`

Sets the `holder` of the receiver to the value of the argument `aHolder`.

`void setNumber(String aNumber)`

Sets the `number` of the receiver to the value of the argument `aNumber`.

`boolean transfer(Account toAccount, double anAmount)`

If the `balance` of the receiver is equal to or greater than the argument `anAmount`, the `balance` of the receiver is debited by the argument `anAmount`. The argument `toAccount` is then credited by the argument `anAmount` and the method returns `true`, otherwise `false` is returned.

Class CurrentAccount

```
java.lang.Object
    └ Account
        └ CurrentAccount
```

```
public class CurrentAccount extends Account
```

The `CurrentAccount` class models simple current accounts, allowing money to be credited to, and debited and transferred from, an account, subject to a given credit limit.

Instance variables

```
private double creditLimit
private String pinNum
```

Constructor Summary

`CurrentAccount()`

Constructor for objects of class `CurrentAccount`. Sets `creditLimit` to `0.0` and `pinNum` to `"0000"`.

```
CurrentAccount(String aHolder, String aNumber,
               double aBalance, double aLimit, String aPin)
```

Constructor for objects of class `CurrentAccount`, which sets the values of the `holder`, `number`, `balance`, `creditLimit` and `pinNum` of the receiver to the arguments `aHolder`, `aNumber`, `aBalance`, `aLimit` and `aPin` respectively.

Method Summary

See also the superclass `Account`.

`double availableToSpend()`

Calculates and returns the amount available to spend (the total of `balance` and `creditLimit`).

`boolean checkPin(String aPin)`

Returns `true` if the `pinNum` of the receiver matches the argument `aPin`, otherwise `false` is returned.

`boolean debit(double anAmount)`

If the amount available to spend (the total of `balance` and `creditLimit`) is equal to or greater than the argument `anAmount`, the `balance` of the receiver is debited by the argument `anAmount` and the method returns `true`, otherwise `false` is returned.

`void displayDetails()`

Prints to the `Display Pane` the `holder`, `number` and `balance` of the receiver.

`boolean equals(CurrentAccount anAccount)`

Returns `true` if receiver is equivalent to (has the same state as) the argument `anAccount`, otherwise `false` is returned.

`double getCreditLimit()`

Returns the `creditLimit` of the receiver.

`String getPinNum()`

Returns the `pinNum` of the receiver.

`void setCreditLimit(double aLimit)`

Sets the `creditLimit` of the receiver to the argument `aLimit`.

`void setPinNum(String aPin)`

Sets the `pinNum` of the receiver to the argument `aPin`.

3.3 Shapes classes

The shape classes were introduced in *Unit 4*, Section 8, and can be seen in Unit4_Project_5_Completed.

Class `Circle`

```
java.lang.Object
    └ java.util.Observable
        └ ou.OUAnimatedObject
            └ Circle
```

`public class Circle extends OUAnimatedObject`

The class `Circle` defines a shape with the characteristics of a circle.

Instance variables

```
private OUColour colour
private int xPos
private int yPos
private int diameter
```

Constructor Summary

`Circle()`

Zero-argument constructor for objects of class `Circle` that sets `colour` to `OUColour.BLUE`, `xPos` to `0` and `yPos` to `0`, and `diameter` to `30`.

`Circle(int aDiameter, OUColour aColour)`

Constructor for objects of class `Circle` with arguments for `diameter` and `colour`, and which sets `xPos` and `yPos` to `0`.

Method Summary

See also the superclass `OUAnimatedObject`.

`OUColour getColour()`

Returns the `colour` of the receiver.

`int getDiameter()`

Returns the `diameter` of the receiver.

`int getXPos()`

Returns the horizontal (`xPos`) position of the receiver.

`int getYPos()`

Returns the vertical position (`yPos`) of the receiver.

`void setColour(OUColour aColour)`

Sets the `colour` of the receiver to the value of the argument `aColour`.

`void setDiameter(int aDiameter)`

Sets the `diameter` of the receiver to the value of the argument `aDiameter`.

`void setXPos(int x)`

Sets the horizontal position (`xPos`) of the receiver to the value of the argument `x`.

`void setYPos(int y)`

Sets the vertical position (`yPos`) of the receiver to the value of the argument `y`.

`String toString()`

Returns a string representation of the receiver.

Class `Diamond`

```
java.lang.Object
    └ java.util.Observable
        └ ou.OUAnimatedObject
            └ Diamond
```

`public class Diamond extends OUAnimatedObject`

The class `Diamond` defines a shape with the characteristics of a diamond.

Instance variables

```
private OUColour colour
private int xPos
private int yPos
private int width
private int height
```

Constructor Summary

`Diamond()`

Zero-argument constructor for objects of class `Diamond` that sets `colour` to `OUColour.GREEN`, `xPos` to `0` and `yPos` to `0`, `width` to `50` and `height` to `100`.

`Diamond(int aWidth, int aHeight, OUColour aColour)`

Constructor for objects of class `Diamond` with arguments for `width`, `height` and `colour`, and which sets `xPos` and `yPos` to `0`.

Method Summary

See also the superclass `OUAnimatedObject`.

`OUColour getColour()`

Returns the `colour` of the receiver.

`int getHeight()`

Returns the `height` of the receiver.

`int getWidth()`

Returns the `width` of the receiver.

`int getXPos()`

Returns the horizontal position (`xPos`) of the receiver.

`int getYPos()`

Returns the vertical position (`yPos`) of the receiver.

`void setColour(OUColour aColour)`

Sets the `colour` of the receiver to the value of the argument `aColour`.

`void setHeight(int aHeight)`

Sets the `height` of the receiver to the value of the argument `aHeight`.

`void setWidth(int aWidth)`

Sets the `width` of the receiver to the value of the argument `aWidth`.

`void setXPos(int x)`

Sets the horizontal position (`xPos`) of the receiver to the value of the argument `x`.

`void setYPos(int y)`

Sets the vertical position (`yPos`) of the receiver to the value of the argument `y`.

`String toString()`

Returns a string representation of the receiver.

Class `Square`

```
java.lang.Object
    └ java.util.Observable
        └ ou.OUAnimatedObject
            └ Square
```

```
public class Square extends OUAnimatedObject
```

The class `Square` defines a shape with the characteristics of a square.

Instance variables

```
private OUColour colour
private int xPos
private int yPos
private int length
```

Constructor Summary

`Square()`

Zero-argument constructor for objects of class `Square` that sets `colour` to `OUColour.ORANGE`, `xPos` to `0` and `yPos` to `0`, and `length` to `15`.

`Square(int aLength, OUColour aColour)`

Constructor for objects of class `Square` with arguments for `length` and `colour`, and which sets both `xPos` and `yPos` to `0`.

Method Summary

See also the superclass `OUAnimatedObject`.

`OUColour getColour()`

Returns the `colour` of the receiver.

`int getLength()`

Returns the `length` of the receiver.

`int getXPos()`

Returns the horizontal position (`xPos`) of the receiver.

`int getYPos()`

Returns the vertical position (`yPos`) of the receiver.

`void setColour(OUColour aColour)`

Sets the `colour` of the receiver to the value of the argument `aColour`.

`void setLength(int aLength)`

Sets the `length` of the receiver to the value of the argument `aLength`.

`void setXPos(int x)`

Sets the horizontal position (`xPos`) of the receiver to the value of the argument `x`.

`void setYPos(int y)`

Sets the vertical position (`yPos`) of the receiver to the value of the argument `y`.

`String toString()`

Returns a string representation of the receiver.

Class `Triangle`

```
java.lang.Object
    ∟ java.util.Observable
        ∟ ou.OUAnimatedObject
            ∟ Triangle
```

`public class Triangle extends OUAnimatedObject`

The class `Triangle` defines a shape with the characteristics of an isosceles triangle.

Instance variables

```
private OUColour colour
private int xPos
private int yPos
private int width
private int height
```

Constructor Summary

`Triangle()`

Zero-argument constructor for objects of class `Triangle` that sets `colour` to `OUColour.RED`, `xPos` to `0`, `yPos` to `0`, `width` to `20` and `height` to `50`.

`Triangle(int aWidth, int aHeight, OUColour aColour)`

Constructor for objects of class `Triangle` with arguments for `width`, `height` and `colour`, and which sets `xPos` and `yPos` to `0`.

Method Summary

See also the superclass `OUAnimatedObject`.

`OUColour getColour()`

Returns the `colour` of the receiver.

`int getHeight()`

Returns the `height` of the receiver.

`int getWidth()`

Returns the `width` of the receiver.

`int getXPos()`

Returns the horizontal position (`xPos`) of the receiver.

`int getYPos()`

Returns the vertical position (`yPos`) of the receiver.

`void setColour(OUColour aColour)`

Sets the `colour` of the receiver to the value of the argument `aColour`.

`void setHeight(int aHeight)`

Sets the `height` of the receiver to the value of the argument `aHeight`.

`void setWidth(int aWidth)`

Sets the `width` of the receiver to the value of the argument `aWidth`.

`void setXPos(int x)`

Sets the horizontal position (`xPos`) of the receiver to the value of the argument `x`.

`void setYPos(int y)`

Sets the vertical position (`yPos`) of the receiver to the value of the argument `y`.

`String toString()`

Returns a string representation of the receiver.

4 Strings

The `String` and `StringBuilder` classes have a number of methods that are very similar and differ only in the types of their arguments. In such cases, we list the method only once and give the argument as `T anArgument`, which is M250 shorthand to tell you that there are multiple methods, one for each of the primitive types and one that takes an argument of type `Object`. For example:

```
static String valueOf(T anArgument)
```

Class `String`

```
java.lang.Object
  └ java.lang.String
```

```
public final class String extends Object
                          implements Comparable<String>
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class. Instances of the `String` class are immutable: they cannot be changed once created.

Constructor summary

`String()`

Initialises a newly created empty `String` object.

`String(char[] value)`

Initialises a new `String` object so that it represents the sequence of characters currently contained in the character array argument.

`String(String original)`

Initialises a newly created `String` object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

`String(StringBuilder builder)`

Initialises a new `String` that contains the sequence of characters currently contained in the `StringBuilder` argument.

Method Summary

In the following methods, where the formal argument is of type `CharSequence`, you can assume for our purposes that the actual argument will be a `String` object.

`char charAt(int index)`

Returns the `char` value at the index specified by the argument.

`int compareTo(String anotherString)`

Compares the receiver to the argument string character by character, based on the Unicode value of each character in the strings.

If the receiver `String` comes before the argument string alphabetically the method returns a negative `int` (note that an upper-case letter comes before the same alphabetical lower-case letter).

If the receiver comes after the argument string alphabetically the method returns a positive `int`.

If the two strings are exactly equal, `0` is returned. The size of the returned `int` (positive or negative) tells you how far apart in the character sequence the first unequal characters are.

`int compareToIgnoreCase(String str)`

Compares the receiver to the argument string character by character as for the `compareTo()` method, but ignoring case differences.

`String concat(String str)`

Returns the receiver if the length of the argument string is `0`. Otherwise, the method returns a new `String` object that is the concatenation of the receiver followed by the argument string.

`boolean contains(CharSequence s)`

Returns `true` if the receiver contains the sequence of `char` values contained in the argument as a subsequence; otherwise returns `false`.

`boolean contentEquals(CharSequence cs)`

Returns `true` if the receiver contains the same sequence of `char` values as the argument; otherwise returns `false`.

`boolean contentEquals(StringBuffer sb)`

Returns `true` if the receiver contains the same sequence of `char` values as the `StringBuffer` argument; otherwise returns `false`.

`static String copyValueOf(char[] data)`

Returns a new `String` which has the same characters, in the same order as the `char` array argument.

`boolean endsWith(String suffix)`

Returns `true` if the receiver ends with the argument; otherwise returns `false`.

`boolean equals(Object anObject)`

Returns `true` if the argument is not `null` and is a `String` object that holds the same sequence of characters as the receiver; otherwise returns `false`.

`boolean equalsIgnoreCase(String anotherString)`

Behaves the same way as the `equals()` method, but ignores the case of the receiver and the argument.

`int hashCode()`

Returns the hash code of the receiver.

In the following `indexOf()` methods, if a formal argument is given as `int ch`, you can assume for our purposes that the actual argument will be of type `char`.

`int indexOf(int ch)`

Returns the index of the first occurrence of the argument `ch` within the receiver. If the character does not occur within the receiver, `-1` is returned.

`int indexOf(int ch, int fromIndex)`

Returns the index of the first occurrence of `ch` within the receiver, starting the search at `fromIndex`. If the character does not occur within the receiver, `-1` is returned.

`int indexOf(String str)`

If the argument occurs as a substring within the receiver, then the index of the first character of the first such substring found is returned; if the argument does not occur as a substring, `-1` is returned.

`int indexOf(String str, int fromIndex)`

If the argument `str` occurs as a substring within the receiver, then the index of the first character of the first such substring found, starting the search at `fromIndex`, is returned; if the argument does not occur as a substring, `-1` is returned.

`int length()`

Returns the length of the receiver.

`String replace(char oldChar, char newChar)`

Returns a copy of the receiver where all occurrences of `oldChar` have been replaced with `newChar`.

`String[] split(String regex)`

Returns an array of strings computed by splitting the receiver around matches of the given regular expression. Any trailing empty strings will be discarded For example:

`"boo:and:foo".split(":");`

returns the array `{"boo", "and", "foo"}`

`"boo:and:foo".split("o");`

returns the array `{"b", "", ":and:f"}`

`boolean startsWith(String prefix)`

Returns `true` if the receiver starts with the argument; otherwise returns `false`.

`String substring(int beginIndex)`

Returns a substring of the receiver beginning at `beginIndex`.

`String substring(int beginIndex, int endIndex)`

Returns a substring of the receiver from `beginIndex` to `endIndex - 1`.

`char[] toCharArray()`

Returns an array of `char` which has the same characters, in the same order as the receiver.

`String toLowerCase()`

Returns a copy of the receiver with all the characters in lower case.

`String toString()`

Returns the receiver, which is already a string!

`String toUpperCase()`

Returns a copy of the receiver with all the characters in upper case.

`String trim()`

Returns a copy of the receiver, with leading and trailing whitespace omitted.

`static String valueOf(T anArgument)`

Returns the string representation of the actual argument which can be of any primitive or reference type.

If `x` references an object then `String.valueOf(x)` and `x.toString()` return the same string. For this reason the `valueOf()` method is mainly used to convert values of primitive types to strings.

Class `StringBuilder`

```
java.lang.Object
   └ java.lang.StringBuilder
```

```
public final class StringBuilder extends Object
                                     implements Appendable
```

The `StringBuilder` class represents character strings. However unlike instances of `String`, instances of `StringBuilder` are mutable, they can be changed once created.

`StringBuilder` implements the `Appendable` interface, which means that it has the `append(char)` method.

Constructor summary

`StringBuilder()`

Initialises a newly created empty `StringBuilder` object.

`StringBuilder(String str)`

Initialises a newly created `StringBuilder` object so that it represents the same sequence of characters as the argument.

Method Summary

`StringBuilder append(T anArgument)`

Appends a string representation of the actual argument (which can be of any primitive or reference type) to the receiver. Method returns the receiver.

`char charAt(int index)`

Returns the `char` value at the index specified by the argument.

`StringBuilder delete(int start, int end)`

Removes a substring from `start` to `end - 1`, from the receiver. Method returns the receiver.

`StringBuilder deleteCharAt(int index)`

Removes the `char` value at the index specified by the argument. Method returns the receiver.

`int indexOf(String str)`

If the argument occurs as a substring within the receiver, then the index of the first character of the first such substring found is returned; if the argument does not occur as a substring, `-1` is returned.

`int indexOf(String str, int fromIndex)`

If the argument `str` occurs as a substring within the receiver, then the index of the first character of the first such substring found, starting the search at `fromIndex`, is returned; if the argument does not occur as a substring, `-1` is returned.

`StringBuilder insert(int offset, T anArgument)`

Inserts a string representation of the second argument (which can be of any primitive or reference type) into the receiver at the index specified by `offset`. Method returns the receiver.

`int length()`

Returns the length of the receiver.

`StringBuilder replace(int start, int end, String str)`

First the characters in a substring of the receiver, which is specified as `start` to `end - 1`, are removed and then the `String` argument is inserted at `start`. Method returns the receiver.

`StringBuilder reverse()`

Reverses the order of the receiver's characters. Method returns the receiver.

`void setCharAt(int index, char ch)`

Replaces the receiver's character at `index` with `ch`.

`String substring(int start)`

Returns a `String` which is a substring of the receiver beginning at `start`.

`String substring(int start, int end)`

Returns a `String` which is a substring of the receiver from `start` to `end - 1`.

`String toString()`

Returns a `String` representation of the receiver.

5 Java Collections Framework

In this section, the type names `E`, `K` and `V` are used as placeholders for real type names that you would use in your code (such as `String`, `Frog`, `Account`, `Integer`, etc.). `E` is used to indicate the declared type of a collection's elements; `K` is used to indicate the declared type of a map's keys; `V` is used to indicate the declared type of a map's values.

We have simplified the argument and return type documentation for this section in two ways. Where standard Javadoc for collections would show the declaration of a formal argument for a method as:

```
void putAll(Map<K, V> aMap)
```

or

```
boolean addAll(Collection<E> aCol)
```

we have simplified them to:

```
void putAll(Map aMap)
```

and

```
boolean addAll(Collection aCol)
```

Similarly for return types. We have simplified

```
SortedMap<K, V> tailMap(K fromKey)
```

to:

```
SortedMap tailMap(K fromKey)
```

5.1 Collection interfaces

Interface `Collection`

`java.util.Collection`

Superinterfaces: `Iterable`

Subinterfaces include: `List`, `Queue`, `Set`, `SortedSet`

Implementing classes include: `ArrayList`, `HashSet`, `LinkedList`, `TreeSet`

```
public interface Collection<E> extends Iterable<E>
```

The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

Method Summary

`boolean add(E element)`

Adds the argument to the receiver. Returns `true` if the operation was successful, `false` otherwise.

`boolean addAll(Collection aCol)`

Adds all of the elements in the argument to the receiver. Returns `true` if the operation was successful, `false` otherwise.

`void clear()`

Removes all of the elements from the receiver (if there were any).

`boolean contains(Object obj)`

Tests whether the argument is present in the receiver. Returns `true` if the argument is an element in the collection, `false` otherwise.

`boolean containsAll(Collection aCol)`

Returns `true` if the receiver contains all of the elements in the argument, `false` otherwise.

`boolean equals(Object obj)`

Compares the argument with the receiver for equality. Returns `true` if the argument object is equal to the receiver, `false` otherwise.

`int hashCode()`

Returns the hash code of the receiver.

`boolean isEmpty()`

Tests whether the receiver is empty. Returns `true` if empty, `false` otherwise.

`boolean remove(Object obj)`

Removes one occurrence of the argument from the collection. Returns `true` if the operation was successful, `false` otherwise.

`boolean removeAll(Collection aCol)`

Removes one occurrence of each of the argument's elements from the receiver. Returns `true` if the operation was successful, `false` otherwise.

`boolean retainAll(Collection aCol)`

Retains *only* the elements in the receiver that are also contained in the argument. Returns `true` if the operation was successful, `false` otherwise.

`int size()`

Returns the number of elements in the receiver.

`Object[] toArray()`

Returns an array containing all of the elements in the receiver.

`E[] toArray(E[] anArray)`

Returns an array containing all the elements in the receiver; the runtime type of the returned array is that of the argument.

Interface `Iterable`

`java.lang.Iterable`

Subinterfaces include: `List`, `Queue`, `Set`, `SortedSet`

Implementing classes include: `ArrayList`, `HashSet`, `LinkedList`, `TreeSet`

`public interface Iterable<E>`

Implementing this interface allows a collection class's instances to be iterated over by a *for-each* statement.

Method Summary

`Iterator<E> iterator()`

Returns an iterator over a collection of elements of type `T`.

See the interface `Iterator` for more information.

Interface `Iterator`

`java.util.Iterator`

Implementing classes include: `Scanner`

`public interface Iterator<E>`

Provides methods that allow iterating over a collection of elements of type `E`.

Method Summary

`boolean hasNext()`

Returns `true` if the iteration has more elements.

`E next()`

Returns the next element in the iteration.

`void remove()`

Optional operation to remove the last element returned by this iterator. (May throw an exception instead of supporting this method.) Iterator behaviour is unspecified if the collection is modified during iteration in any way other than by calling this method.

Interface `List`

`java.util.List`

Superinterfaces: `Collection`, `Iterable`

Implementing classes include: `ArrayList`, `LinkedList`

`public interface List<E> extends Collection<E>`

An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. Unlike sets, lists allow duplicate elements.

Method Summary

See also the superinterface `Collection`.

`boolean add(E element)`

Appends the argument to the end of the receiver. Returns `true` if the operation was successful, `false` otherwise.

`void add(int index, E element)`

Inserts the argument (`element`) into the receiver at the position specified by the argument `index`. The method shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

`boolean addAll(Collection aCol)`

Appends all of the elements in the argument to the end of the receiver, in the order that they are returned by the argument's iterator. Returns `true` if the operation was successful, `false` otherwise.

`boolean addAll(int index, Collection aCol)`

Inserts all of the elements in the argument `aCol` into the receiver at the position specified by the argument `index`, shifting any existing elements from this index onwards to the right. Returns `true` if the operation was successful, `false` otherwise.

`boolean equals(Object obj)`

Compares the argument `obj` with the receiver for equality. Returns `true` if, and only if, the argument is also a list, both the receiver and the argument have the same size, and all corresponding pairs of elements in the two lists are equal, `false` otherwise.

`E get(int index)`

Returns the element at the position specified by the argument.

`int hashCode()`

Returns the hash code for the receiver.

`int indexOf(Object obj)`

Returns the index in the receiver of the first occurrence of the argument, or `-1` if the receiver does not contain this element.

`int lastIndexOf(Object obj)`

Returns the index in the receiver of the last occurrence of the argument, or `-1` if the receiver does not contain this element.

`E remove(int index)`

Removes and returns the element at the position specified by the argument.

`boolean remove(Object obj)`

Removes the first occurrence in the receiver of the argument. Returns `true` if the operation was successful, `false` otherwise.

`E set(int index, E element)`

Replaces the element at the position specified by the first argument with the second argument and returns the original element.

`int size()`

Returns the number of elements in the receiver.

`List subList(int fromIndex, int toIndex)`

Returns a view of a portion of the receiver between the indices specified by the arguments `fromIndex` (inclusive), and `toIndex` (exclusive). Changes in the returned list are reflected in the original list, and vice-versa.

`Object[] toArray()`

Returns an array containing all of the elements in the receiver in the same order.

`E[] toArray(E[] anArray)`

Returns an array containing all the elements in the receiver in the same order; the run-time type of the returned array is that of the argument.

Interface `Set`

`java.util.Set`

Superinterfaces: `Collection`, `Iterable`

Subinterfaces include: `SortedSet`

Implementing classes include: `HashSet`, `TreeSet`

`public interface Set<E> extends Collection<E>`

A collection that contains no duplicate elements, and at most one null element. As implied by its name, this interface models the mathematical *set* abstraction.

Method Summary

See also the superinterfaces `Collection` and `Iterable`.

`boolean add(E element)`

Adds the argument to the receiver, unless it is already there. Returns `true` if the argument was added, `false` if it was not.

`boolean addAll(Collection aCol)`

Adds all of the elements in the argument to the receiver if they are not already present. Returns `true` if the operation was successful, `false` otherwise.

`boolean equals(Object obj)`

Compares the argument with the receiver for equality. Returns `true` if the argument is also a set, the two sets have the same size, and every element in the argument is also contained in the receiver, `false` otherwise.

`int hashCode()`

Returns the hash code of the receiver.

Interface `SortedSet`

`java.util.SortedSet`

Superinterfaces: `Collection`, `Iterable`, `Set`

Implementing classes include: `TreeSet`

`public interface SortedSet<E> extends Set<E>`

A set that keeps its elements ordered according to their natural ordering. Several additional operations are provided to take advantage of this ordering.

Method Summary

See also the superinterfaces `Collection`, `Set` *and* `Iterable`.

`E first()`

Returns the first (lowest) element currently in the receiver.

`SortedSet headSet(E toElement)`

Returns a view of the portion of the receiver whose elements are strictly less than `toElement`. Changes in the returned `SortedSet` are reflected in the original set, and vice-versa.

`E last()`

Returns the last (highest) element currently in the receiver.

`SortedSet subSet(E fromElement, E toElement)`

Returns a view of a portion of the receiver between the elements specified by the arguments `fromElement` (inclusive), and `toElement` (exclusive). Changes in the returned `SortedSet` are reflected in the original set, and vice-versa.

`SortedSet tailSet(E fromElement)`

Returns a view of the portion of the receiver whose elements are greater than or equal to `fromElement`. Changes in the returned `SortedSet` are reflected in the original set, and vice-versa.

Interface `Map`

`java.util.Map`

Subinterfaces include: `SortedMap`

Implementing classes include: `HashMap`, `TreeMap`

`public interface Map<K,V>`

A collection that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

Method Summary

`void clear()`

Removes all entries from the receiver.

`boolean containsKey(Object key)`

Tests whether a given key is present in the receiver. The method returns `true` if the key is present, `false` otherwise.

`boolean containsValue(Object value)`

Tests whether the argument is present as a value in the receiver. The method returns `true` if the argument is present as a value, `false` otherwise.

`boolean equals(Object obj)`

Compares the argument with the receiver for equality. Returns `true` if the argument is also a map and the two maps represent the same key–value pairs.

`V get(Object key)`

If the argument exists as a key in the receiver, then the method returns the associated value. Otherwise `null` is returned.

`int hashCode()`

Returns the hash code of the receiver.

`boolean isEmpty()`

Tests whether the receiver contains any key–value pairs. The method returns `true` if the map is empty, otherwise `false`.

`Set keySet()`

Returns a view of the keys contained in the receiver as a set. If a key is removed from the returned set, then the associated key–value pair is removed from the map and vice versa.

`V put(K key, V value)`

Puts a key–value pair into the receiver. If a key–value pair already exists with the same key, then the old value is overwritten by the new value. The method returns the previous value for the key, if there was one, otherwise it returns `null`.

`void putAll(Map aMap)`

Copies all of the key–value pairs from the argument into the receiver.

`V remove(Object key)`

Removes the key–value pair associated with the argument (if the key exists). Returns the previous value for the key, if there was one, otherwise `null`.

`int size()`

Returns the number of key–value mappings in the receiver.

`Collection values()`

Returns a view of the values contained in receiver as a collection. As this collection is just a view of the values in the map, if a value is removed from the returned collection, then the associated key–value pair is removed from the map.

Interface `SortedMap`

`java.util.SortedMap`

Superinterfaces: `Map`

Implementing classes include: `TreeMap`

```
public interface SortedMap<K,V> extends Map<K,V>
```

A map that guarantees that its elements will be in ascending key order, sorted according to the natural ordering of its keys.

Method Summary

See also the superinterface `Map`.

`K firstKey()`

Returns the first (lowest) key currently in the receiver.

`SortedMap headMap(K toKey)`

Returns a view of the portion of the receiver whose keys are strictly less than `toKey`.

`K lastKey()`

Returns the last (highest) key currently in the receiver.

`SortedMap subMap(K fromKey, K toKey)`

Returns a view of the portion of the receiver whose keys range from `fromKey` (inclusive) to `toKey` (exclusive).

`SortedMap tailMap(K fromKey)`

Returns a view of the portion of the receiver whose keys are greater than or equal to `fromKey`.

5.2 Collection implementation classes

Class `HashSet`

```
java.lang.Object
   └ java.util.AbstractCollection
        └ java.util.AbstractSet
             └ java.util.HashSet
```

```
public class HashSet<E> extends AbstractSet<E>
          implements Collection<E>, Iterable<E>, Set<E>
```

This class implements the `Set` interface.

Constructor Summary

`HashSet()`

Constructs an empty set.

`HashSet(Collection aCol)`

Constructs a set containing the elements in the argument.

Method Summary

See the interfaces `Set`, `Collection` and `Iterable`.

Class `TreeSet`

```
java.lang.Object
   └ java.util.AbstractCollection
      └ java.util.AbstractSet
         └ java.util.TreeSet
```

```
public class TreeSet<E> extends AbstractSet<E>
     implements Collection<E>, Iterable<E>, SortedSet<E>
```

This class implements the `SortedSet` interface. The class guarantees that the sorted set will be in ascending element order, sorted according to the natural order of the elements.

Constructor Summary

`TreeSet()`

Constructs an empty `TreeSet`.

`TreeSet(Collection aCol)`

Constructs a `TreeSet` containing the elements in the argument, sorted according to the elements' natural order.

`TreeSet(SortedSet aSortedSet)`

Constructs a `TreeSet` containing the same elements as the `SortedSet` given by the argument.

Method Summary

See the interfaces `Collection`, `Set`, `SortedSet` and `Iterable`.

Class `HashMap`

```
java.lang.Object
   └ java.util.AbstractMap
      └ java.util.HashMap
```

```
public class HashMap<K,V> extends AbstractMap<K,V>
                              implements Map<K,V>
```

Implementation of the `Map` interface.

Constructor Summary

`HashMap()`

Constructs an empty `HashMap`.

`HashMap(Map aMap)`

Constructs a `HashMap` with the same mappings as the argument.

Method Summary

See the interface `Map`.

Class `TreeMap`

```
java.lang.Object
   └ java.util.AbstractMap
      └ java.util.TreeMap
```

```
public class TreeMap<K,V> extends AbstractMap<K,V>
                          implements SortedMap<K,V>
```

Implementation of the `SortedMap` interface. This class guarantees that the map will be sorted in ascending key order, according to the natural order of the map's keys.

Constructor Summary

`TreeMap()`

Constructs an empty `TreeMap`.

`TreeMap(Map aMap)`

Constructs a `TreeMap` containing the same mappings as the argument, sorted according to the keys' natural order.

`TreeMap(SortedMap aSortedMap)`

Constructs a `TreeMap` containing the same mappings as the argument, sorted according to the same ordering.

Method Summary

See the interfaces `Map` and `SortedMap`.

Class `ArrayList`

```
java.lang.Object
   └java.util.AbstractCollection
      └java.util.AbstractList
         └java.util.ArrayList
```

```
public class ArrayList<E> extends AbstractList<E>
       implements Collection<E>, List<E>, Iterable<E>
```

Implementation of the `List` interface.

Constructor Summary

`ArrayList()`

Constructs an empty `ArrayList`.

`ArrayList(Collection aCol)`

Constructs an `ArrayList` containing the elements of the argument, in the order they appear in the argument.

Method Summary

See the interfaces `List`, `Collection` and `Iterable`.

5.3 Collection utility classes

Class `Arrays`

```
java.lang.Object
    └ java.util.Arrays
```

```
public class Arrays extends Object
```

A utility class that contains static methods for manipulating arrays (such as sorting and searching). The methods in this class throw a `NullPointerException` if the specified array reference is `null`.

Method Summary

`static List asList(E[] anArray)`

Returns a fixed-size list backed by the array argument. (Changes to the returned list are reflected in the array and vice versa.)

`static int binarySearch(int[] intArray, int anInt)`

Searches the array argument for the value specified by the second argument using the binary search algorithm. If found the method returns the index of the element, otherwise it returns (-(insertion point) - 1).

The array being searched must be sorted (as by the `sort(int)` method, below), if not, the results are undefined. If the array contains multiple elements with the same specified value, there is no guarantee which one will be found.

There are equivalent methods for the primitive types `byte`, `char`, `double`, `float`, `long`, `short`.

`static int binarySearch(Object[] anArray, Object obj)`

Searches the array argument for the object specified by the second argument using the binary search algorithm. If found the method returns the index of the element, otherwise it returns (-(insertion point) - 1).

The array being searched must be sorted into ascending order according to the natural ordering of its elements (as by the `sort(Object[]` method, below), if not, the results are undefined. If the array contains multiple elements equal to the specified object, there is no guarantee which one will be found.

```
static boolean deepEquals(Object[] anArray1,
                          Object[] anArray2)
```

Returns `true` if the two array arguments are *deeply equal* to one another. Two array references are considered deeply equal if both are `null`, or if they refer to arrays that contain the same number of elements and all corresponding pairs of elements in the two arrays are deeply equal.

`static int deepHashCode(Object[] anArray)`

Returns a hash code based on the ‘deep contents’ of the array argument. If the array contains other arrays as elements, the hash code is based on their contents and so on, *ad infinitum*.

`static String deepToString(Object[] anArray)`

Returns a string representation of the 'deep contents' of the array argument. If the array contains other arrays as elements, the string representation contains their contents and so on.

`static boolean equals(int[] anArray, int[] anArray2)`

Returns `true` if the two array arguments are equal to one another, `false` otherwise. There are equivalent methods for the primitive types `boolean`, `byte`, `char`, `double`, `float`, `long`, `short`.

```
static boolean equals(Object[] anArray,
                      Object[] anArray2)
```

Returns `true` if the two array arguments are equal to one another, `false` otherwise.

`static void fill(int[] anArray, int val)`

Assigns the `int` argument `val` to each component of the argument array. There are equivalent methods for the primitive types `boolean`, `byte`, `char`, `double`, `float`, `long`, `short`.

```
static void fill(int[] anArray, int fromIndex,
                 int toIndex, int val)
```

Assigns the `int` argument `val` to the components of a sub-array of the array argument. The range of the sub-array is given by the arguments `fromIndex` (inclusive) and `toIndex` (exclusive). There are equivalent methods for the primitive types `boolean`, `byte`, `char`, `double`, `float`, `long`, `short`.

```
static void fill(Object[] anArray, int fromIndex,
                 int toIndex, Object val)
```

Assigns the `Object` argument `val` to the components of a sub-array of the array argument. The range of the sub-array is given by the arguments `fromIndex` (inclusive) and `toIndex` (exclusive).

`static void fill(Object[] anArray, Object val)`

Assigns the `Object` argument `val` to each component of the array argument.

`static int hashCode(int[] anArray)`

Returns a hash code based on the contents of the array argument. There are equivalent methods for the primitive types `boolean`, `byte`, `char`, `double`, `float`, `long`, `short`.

`static int hashCode(Object[] anArray)`

Returns a hash code based on the contents of the array argument.

`static void sort(int[] anArray)`

Sorts the argument, an array of integers into ascending numerical order. There are equivalent methods for the primitive types: `byte`, `char`, `double`, `float`, `long`, `short`.

`static void sort(int[] anArray, int fromIndex, int toIndex)`

Sorts a sub-array of the array argument, into ascending numerical order. The sub-array is specified by the arguments `fromIndex` (inclusive) and `toIndex` (exclusive). There are equivalent methods for the primitive types: `byte`, `char`, `double`, `float`, `long`, `short`.

`static void sort(Object[] anArray)`

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements.

`static void sort(Object[] anArray, int fromIndex, int toIndex)`

Sorts a sub-array of the array argument into ascending order, according to the natural ordering of its elements. The sub-array is specified by the arguments `fromIndex` (inclusive) and `toIndex` (exclusive).

`static String toString(int[] anArray)`

Returns a string representation of the contents of the argument array. There are equivalent methods for the primitive types `boolean`, `byte`, `char`, `double`, `float`, `long`, `short`.

`static String toString(Object[] anArray)`

Returns a string representation of the contents of the argument array.

Class `Collections`

```
java.lang.Object
   └ java.util.Collections
```

`public class Collections extends Object`

A utility class consisting exclusively of static methods that operate on, or return, collections.

Method Summary

`static boolean disjoint(Collection col1, Collection col2)`

Returns `true` if the two arguments have no elements in common, `false` otherwise.

`static int frequency(Collection aCol, Object obj)`

Returns the number of elements in the argument `aCol` that are equal to the argument `obj`.

`static int indexOfSubList(List source, List subList)`

If `sublist` is a sub-list of `source` the method returns the index of the first element of `subList` within `source`, otherwise `-1` is returned.

```
static int lastIndexOfSubList(List source,
                              List subList)
```

If `sublist` is a sub-list of `source` the method returns the index of the last element of `subList` within `source`, otherwise `-1` is returned.

```
static E max(Collection coll)
```

Returns the maximum element in the argument, according to the natural ordering of its elements. All elements in the collection must implement the `Comparable` interface.

```
static E min(Collection coll)
```

Returns the minimum element in the argument, according to the natural ordering of its elements. All elements in the collection must implement the `Comparable` interface.

```
static void reverse(List list)
```

Reverses the order of the elements in the argument.

```
static void rotate(List list, int distance)
```

Rotates the elements in the specified list by the specified `distance`.

```
static void shuffle(List list)
```

Randomly shuffles the arguments elements.

```
static void sort(List list)
```

Sorts the argument into ascending order, according to the natural ordering of its elements. All elements in the list must implement the `Comparable` interface.

```
static void swap(List list, int i, int j)
```

Swaps the elements of the argument that are at indices `i` and `j`.

6 Files and streams

This section contains classes for writing to and reading from files.

6.1 Files and pathnames

Class `File`

```
java.lang.Object
   └ java.io.File
```

```
public class File extends Object
                        implements Comparable<File>
```

An abstract representation of file and directory pathnames.

Constructor Summary

`File(String pathname)`

Constructs a `File` instance by converting the given pathname string into an abstract pathname.

Method Summary

`boolean canRead()`

Returns `true` if the application can read the file denoted by the receiver, otherwise `false`.

`boolean canWrite()`

Returns `true` if the application can modify the file denoted by the receiver, otherwise `false`.

`boolean exists()`

Returns `true` if the file or directory denoted by the receiver exists, otherwise `false`.

`boolean isDirectory()`

Returns `true` if the file denoted by the receiver is a directory (folder), otherwise `false`.

`boolean isFile()`

Returns `true` if the file denoted by the receiver is a normal file, otherwise `false`.

`String toString()`

Returns the pathname string of the receiver.

6.2 Reading from character streams

These classes are used for reading character data from input streams. They implement the interface `Readable`, which means that they have a method to read from a character buffer and the interface `Closeable`, which means that they have the `close()` method.

Class `Reader`

```
java.lang.Object
    └ java.io.Reader
```

```
public abstract class Reader extends Object
                                implements Readable, Closeable
```

Abstract class for reading character streams. Instances of subclasses of the `Reader` class handle (16 bit) character streams; this means that they correctly handle textual information based on characters and strings.

Method Summary

`abstract void close() throws IOException`

Closes the stream.

`int read() throws IOException`

Reads a single character. Returns the character read, as an integer in the range 0 to 65535, or `-1` if the end of the stream has been reached.

`int read(char[] cbuf) throws IOException`

Reads characters into an array specified by the argument `cbuf`. Returns the number of characters read, or `-1` if the end of the stream has been reached.

```
abstract int read(char[] cbuf, int offSet, int length)
                        throws IOException
```

Reads characters into a portion of `cbuf` storing the first character at `offSet` and reading `length` characters. Returns the number of characters read, or `-1` if the end of the stream has been reached.

`int read(CharBuffer target) throws IOException`

Attempts to read characters into the specified character buffer `target`. Returns the number of characters added to the buffer, or `-1` if this source of characters is at its end.

`long skip(long n) throws IOException`

Skips `n` characters.

Class `FileReader`

```
java.lang.Object
    └ java.io.Reader
        └ java.io.InputStreamReader
            └ java.io.FileReader
```

```
public class FileReader extends InputStreamReader
                implements Readable, Closeable
```

The simplest `Reader` subclass to use to open an input stream to read characters from a text file.

Constructor Summary

`FileReader(File file) throws FileNotFoundException`

Constructs a new `FileReader`, given the `File` to read from.

Method Summary

See the `Reader` abstract class.

Class `BufferedReader`

```
java.lang.Object
    └ java.io.Reader
        └ java.io.BufferedReader
```

```
public class BufferedReader extends Reader
                implements Readable, Closeable
```

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays and lines. In general, each read request made of a `Reader` causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to use a `BufferedReader` to wrap any instance of a subclass of `Reader` (such as instances of `FileReader`) whose `read()` operations may be costly.

Constructor Summary

`BufferedReader(Reader in)`

Constructs a buffered character-input stream.

Method Summary

See also the `Reader` abstract class.

`String readLine() throws IOException`

Reads a line of text. A line is considered to be terminated by any one of a linefeed (`'\n'`), a carriage return (`'\r'`), or a carriage return followed immediately by a linefeed. Returns a `String` containing the contents of the line, not including any line-termination characters, or `null` if the end of the stream has been reached.

Class `Scanner`

```
java.lang.Object
    └ java.util.Scanner
```

```
public final class Scanner extends Object
                implements Closeable, Iterator<String>
```

A simple text scanner which can parse primitive types and strings using regular expressions. A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various `next` methods.

Constructor Summary

`Scanner(File source) throws FileNotFoundException`

Constructs a `Scanner` that produces values scanned from the specified file.

`Scanner(Readable source)`

Constructs a `Scanner` that produces values scanned from the specified source. Note that the `Reader` class implements the `Readable` interface so subclasses of `Reader` can be wrapped by a scanner.

`Scanner(String source)`

Constructs a `Scanner` that produces values scanned from the specified string.

Method Summary

`void close()`

Closes the receiver.

`Pattern delimiter()`

Returns the `Pattern` the receiver is currently using to match delimiters.

`boolean hasNext()`

Returns `true` if the receiver has another token in its input.

`boolean hasNextInt()`

Returns `true` if the next token in the receiver's input can be interpreted as an `int` value, `false` otherwise.

The following similar methods are also available: `hasNextBigDecimal()`, `hasNextBigInteger()`, `hasNextBoolean()`, `hasNextByte()`, `hasNextDouble()`, `hasNextFloat()`, `hasNextLong()`, `hasNextShort()`

`boolean hasNextLine()`

Returns `true` if there is another line in the input of the receiver.

`String next()`

Finds and returns the next complete token from the receiver.

`int nextInt()`

Scans and returns the next token of the input as an `int`.

In addition the following similar methods are also available: `nextBigDecimal()`, `nextBigInteger()`, `nextBoolean()`, `nextByte()`, `nextDouble()`, `nextFloat()`, `nextLong()`, `nextShort()`

`String nextLine()`

Advances the receiver past the current line and returns the input that was skipped as a string.

`Scanner useDelimiter(String pattern)`

Sets the receiver's delimiting pattern to a pattern constructed from the specified `String`. Returns this scanner.

6.3 Writing to character streams

These classes are used for writing character data to output streams. They implement the interface `Appendable`, which means that they have the `append(char)` method; the interface `Closeable`, which means that they have the `close()` method, and the interface `Flushable`, which means that they have the `flush()` method.

Class `Writer`

```
java.lang.Object
    └ java.io.Writer
```

```
public abstract class Writer extends Object
                implements Appendable, Closeable, Flushable
```

Abstract class for writing to character streams. Instances of subclasses of the `Writer` class handle (16 bit) character streams; this means that they correctly handle textual information based on characters and strings.

Method Summary

In the following methods, where the formal argument is of type `CharSequence` *you can assume for our purposes that the actual argument will be a* `String` *object or a* `StringBuider` *object.*

`Writer append(char c) throws IOException`

Appends the argument character `c` to the receiver. Returns the receiver.

`Writer append(CharSequence csq) throws IOException`

Appends the argument character sequence `csq` to the receiver. Returns the receiver.

```
Writer append(CharSequence csq, int start, int end)
             throws IOException
```

Appends a subsequence of the argument character sequence `csq` to the receiver, where `start` is the index of the first character in the subsequence and `end` is the index of the character following the last character. Returns the receiver.

`abstract void close() throws IOException`

Closes the stream, flushing it first.

`abstract void flush() throws IOException`

Flushes the stream. If the stream has saved any characters from the various `write()` methods in a buffer, they are written immediately to their intended destination. Then, if that destination is another character or byte stream it is flushed. Thus one `flush()` invocation will flush all the buffers in a chain of `Writers` and `OutputStreams`. If the intended destination of this stream is an abstraction provided by the underlying operating system, for example a file, then flushing the stream guarantees only that bytes previously written to the stream are passed to the operating system for writing; it does not guarantee that they are actually written to a physical device such as a disk drive.

```
void write(char[] cbuf) throws IOException
```

Writes the argument array of characters `cbuf`.

```
abstract void write(char[] cbuf, int offSet, int length)
                      throws IOException
```

Writes `length` characters of the array `cbuf` starting with the character in index position `offSet`.

```
void write(int c) throws IOException
```

Writes a single character.

```
void write(String str) throws IOException
```

Writes a string.

```
void write(String str, int off, int len)
            throws IOException
```

Writes `length` characters of the string `str` starting with the character in index position `offSet`.

Class `FileWriter`

```
java.lang.Object
    └ java.io.Writer
        └ java.io.OutputStreamWriter
            └ java.io.FileWriter
```

```
public class FileWriter extends OutputStreamWriter
                implements Appendable, Closeable, Flushable
```

The simplest `Writer` subclass to use to open an output stream to write characters to a text file.

Constructor Summary

```
FileWriter(File file) throws IOException
```

Constructs a `FileWriter` object given a `File` object. Anything written to `file` will be added at the beginning, so overwriting any existing contents.

```
FileWriter(File file boolean append)
            throws IOException
```

Constructs a `FileWriter` object given a `File` object. If `append` is `true` then anything written to `file` will be added at the end. If `append` is `false`, then anything written to `file` will be added at the beginning, so overwriting any existing contents.

Method Summary

See the `Writer` abstract class.

Class `BufferedWriter`

```
java.lang.Object
    └ java.io.Writer
        └ java.io.BufferedWriter
```

```
public class BufferedWriter extends Writer
              implements Appendable, Closeable, Flushable
```

Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays and strings. It is advisable to wrap a `BufferedWriter` around any `Writer` (such as instances of `FileWriter`) whose `write()` operations may be costly.

Constructor Summary

`BufferedWriter(Writer out)`

Constructs a buffered character-output stream that uses a default-sized output buffer.

Method Summary

See also the `Writer` abstract class.

`void newLine() throws IOException`

Writes a line separator. This method uses the platform's own notion of line separator as defined by the system property `line.separator`. Using this method to terminate each output line is therefore preferred to writing a newline character directly.

7 Exceptions

The most important examples of errors and exception classes are listed in this section. All of these classes have a zero-argument constructor as well as a constructor taking a `String` as an argument (which is used as the error or exception message).

Class `Throwable`

```
java.lang.Object
    └ java.lang.Throwable
```

```
public class Throwable extends Object
```

The `Throwable` class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java `throw` statement. Similarly, only this class or one of its subclasses can be the argument type in a `catch` clause.

Method Summary

`Throwable getCause()`

Returns the cause of this throwable or `null` if the cause is nonexistent or unknown.

`String getMessage()`

Returns the detailed message string of this throwable.

`void printStackTrace()`

Displays information about this throwable and its stack trace.

`String toString()`

Returns a short description of this throwable.

7.1 Checked exceptions

Class `Exception`

```
java.lang.Object
    └ java.lang.Throwable
        └ java.lang.Exception
```

```
public class Exception extends Throwable
```

The class `Exception` and its subclasses are a form of `Throwable` that indicates conditions that a reasonable application might want to catch. All direct subclasses of `Exception` (except `RuntimeException` and its subclasses) are checked exceptions.

Method Summary

See the `Throwable` class.

Class `FileNotFoundException`

```
java.lang.Object
    └ java.lang.Throwable
        └ java.lang.Exception
            └ java.io.IOException
                └ java.io.FileNotFoundException
```

```
public class FileNotFoundException extends IOException
```

Signals that an attempt to open the file denoted by a specified pathname has failed.

Method Summary

See the `Throwable` *class.*

Class `IOException`

```
java.lang.Object
    └ java.lang.Throwable
        └ java.lang.Exception
            └ java.io.IOException
```

```
public class IOException extends Exception
```

Signals that an I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations.

Method Summary

See the `Throwable` *class.*

7.2 Unchecked exceptions

Class `Error`

```
java.lang.Object
    └java.lang.Throwable
        └java.lang.Error
```

```
public class Error extends Throwable
```

`Error` is the superclass of those exceptions that should not be caught because they indicate abnormal conditions.

Method Summary

See the `Throwable` *class.*

Class `AssertionError`

```
java.lang.Object
    └ java.lang.Throwable
        └ java.lang.Error
            └ java.lang.AssertionError

public class AssertionError extends Error
```

This class of error is signalled to indicate that an assertion has failed.

Method Summary

See the `Throwable` *class.*

Class `RuntimeException`

```
java.lang.Object
    └java.lang.Throwable
        └java.lang.Exception
            └java.lang.RuntimeException

public class RuntimeException extends Exception
```

`RuntimeException` is the superclass of those unchecked exceptions that need not be caught or declared to be thrown. A method is not required to catch any instances of subclasses of `RuntimeException` that might be thrown during the execution of that method as these exceptions are unchecked exceptions.

Method Summary

See the `Throwable` *class.*

Class `ArrayIndexOutOfBoundsException`

```
java.lang.Object
 └ java.lang.Throwable
    └ java.lang.Exception
       └ java.lang.RuntimeException
          └ java.lang.IndexOutOfBoundsException
             └ java.lang.ArrayIndexOutOfBoundsException

public class ArrayIndexOutOfBoundsException
                        extends IndexOutOfBoundsException
```

Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

Method Summary

See the `Throwable` *class.*

Class `ArithmeticException`

```
java.lang.Object
    └ java.lang.Throwable
        └ java.lang.Exception
            └ java.lang.RuntimeException
                └java.lang.ArithmeticException
```

```
public class ArithmeticException extends RuntimeException
```

Thrown when an exceptional arithmetic condition has occurred. For example, an integer 'divide by zero' throws an instance of this class.

Method Summary

See the `Throwable` class.

Class `IllegalArgumentException`

```
java.lang.Object
    └ java.lang.Throwable
        └ java.lang.Exception
            └ java.lang.RuntimeException
                └ java.lang.IllegalArgumentException
```

```
public class IllegalArgumentException
                        extends RuntimeException
```

Thrown to indicate that a method has been passed an illegal or inappropriate argument.

Method Summary

See the `Throwable` class.

Class `NullPointerException`

```
java.lang.Object
    └ java.lang.Throwable
        └ java.lang.Exception
            └ java.lang.RuntimeException
                └ java.lang.NullPointerException
```

```
public class NullPointerException extends RuntimeException
```

Thrown when an application attempts to use `null` in a case where an object is required.

Method Summary

See the `Throwable` class.

Class `NumberFormatException`

```
java.lang.Object
    └ java.lang.Throwable
        └ java.lang.Exception
            └ java.lang.RuntimeException
                └ java.lang.IllegalArgumentException
                    └ java.lang.NumberFormatException
```

```
public class NumberFormatException
               extends IllegalArgumentException
```

Thrown to indicate that the executing method has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.

Method Summary

See the `Throwable` class.

Class `StringIndexOutOfBoundsException`

```
java.lang.Object
 └ java.lang.Throwable
    └ java.lang.Exception
        └ java.lang.RuntimeException
           └ java.lang.IndexOutOfBoundsException
              └ java.lang.StringIndexOutOfBoundsException
```

```
public class StringIndexOutOfBoundsException
                     extends IndexOutOfBoundsException
```

Thrown to indicate that a string has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the string.

Method Summary

See the `Throwable` class.

8 Java operators used in M250

The following is a table of the Java operators used in M250, shown in order of precedence – from highest to lowest.

Priority	Operators	Precedence
1	postfix	`++` `--`
2	unary	`-` `!`
3	multiplicative	`*` `/` `%`
4	additive	`+` `-`
5	relational	`<` `>` `<=` `>=` `instanceof`
6	equality	`==` `!=`
7	logical AND	`&&`
8	logical OR	`||`
9	assignment	`=`

9 Java keywords

The following table includes all the keywords in the Java language.

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Index

Classes shown in **bold**, methods shown in `code style`